A Multilabel Classification Framework for Approximate Nearest Neighbor Search

Ville Hyvönen
Department of Computer Science
University of Helsinki
Helsinki Institute for Information Technology (HIIT)
ville.o.hyvonen@helsinki.fi

Elias Jääsaari
Machine Learning Department
Carnegie Mellon University
ejaeaesa@andrew.cmu.edu

Teemu Roos
Department of Computer Science
University of Helsinki
Helsinki Institute for Information Technology (HIIT)
teemu.roos@cs.helsinki.fi

Abstract

Both supervised and unsupervised machine learning algorithms have been used to learn partition-based index structures for approximate nearest neighbor (ANN) search. Existing supervised algorithms formulate the learning task as finding a partition in which the nearest neighbors of a training set point belong to the same partition element as the point itself, so that the nearest neighbor candidates can be retrieved by naive lookup or backtracking search. We formulate candidate set selection in ANN search directly as a multilabel classification problem where the labels correspond to the nearest neighbors of the query point, and interpret the partitions as partitioning classifiers for solving this task. Empirical results suggest that the natural classifier based on this interpretation leads to strictly improved performance when combined with any unsupervised or supervised partitioning strategy. We also prove a sufficient condition for consistency of a partitioning classifier for ANN search, and illustrate the result by verifying this condition for chronological k-d trees.

1 Introduction

Approximate nearest neighbor (ANN) search is a fundamental algorithmic problem. There is a large body of literature on ANN search spanning several research communities, including the machine learning community. Specifically, space-partitioning data structures—such as space-partitioning trees (Friedman et al., 1976; Muja and Lowe, 2014; Dasgupta and Sinha, 2015) and data-dependent hash tables (Indyk and Motwani, 1998; Datar et al., 2004; Weiss et al., 2009)—are machine learning methods commonly used for ANN search.

In this article, we propose an intuitive theoretical framework for partition-based ANN search. In particular, we formulate the candidate set selection directly as a multilabel classification problem where the labels represent the indices of the nearest neighbors of the query point. This formulation suggests that the performance of space-partitioning data structures can be improved by using them in a theoretically justified fashion as partitioning classifiers (Devroye et al., 1996, Chapter 21) instead of searching them under the earlier lookup-based paradigm. Our classification framework also enables applying general purpose classifiers—such as a multilabel random forest—directly as an index structure for ANN search.

36th Conference on Neural Information Processing Systems (NeurIPS 2022).

We start by reviewing the relevant background on ANN search and multilabel classification (Sec. 2). Then we formulate candidate set selection in ANN search as a multilabel classification task (Sec. 3). In Sec. 4 we define the natural (partitioning) classifier for the general multilabel classification task. In Sec. 6 we show that interpreting the earlier lookup-based candidate set selection methods in the multilabel classification framework of Sec. 3 suggests that they define a sub-optimal classifier. Our multilabel formulation also enables us to consider asymptotics in the standard statistical learning framework: we establish a sufficient condition for consistency of a partitioning classifier for ANN search (Sec. 7.1). As a concrete example (Sec. 7.2), we verify this condition for the chronological k-d tree (Bentley, 1975) that was the first data structure proposed for accelerating nearest neighbor search. To empirically validate the proposed framework, we show that using a natural classifier that is aligned with the ANN task in conjunction with space-partitioning data structures proposed in the literature leads to strictly improved empirical performance compared to the earlier lookup-based candidate set selection methods (Sec. 8).

2 Background and notation

2.1 Approximate nearest neighbor search

Let the corpus points $\{c_j\}_{j=1}^m$ and the query point x be vectors in $\mathbb{R}^d$. We call the k corpus points that are closest[1] to the query point x its k *nearest neighbors* and denote the set of their indices by

$$\text{NN}_k(x) := k\text{−}\underset{j=1,\ldots,m}{\operatorname{argmin}} \|x - c_j\|, \tag{1}$$

where the notation $k\text{−}\operatorname{argmin} f$ means the set of k values for which the function f has the smallest values, and $\|\cdot\|$ is the Euclidean distance. Other metrics, or—more generally—dissimilarity measures can also be used to define the nearest neighbors.

The trivial solution to the problem finding the nearest neighbors $\text{NN}_k(x)$ of the query point x is to compute the distances to all the corpus point and sort these distances. However, when the dimensionality of the data is high and the corpus size is large, this brute force solution is often too slow if the application requires fast response times. The first data structure to proposed for speeding up nearest neighbor search was a k-d tree (Bentley, 1975). However, for high-dimensional data a k-d tree is not faster than the brute force approach for exact nearest neighbor search because of the well-known "curse of dimensionality" that affects the non-parametric statistical methods—including partitioning methods for nearest neighbor search—in general (Lee and Wong, 1977). Although the query speed of index structures for *exact* nearest neighbor search often degrades when the dimensionality of the data increases so that they are not an improvement on the brute force approach, this problem can be mitigated by allowing an approximate solution. This is why in modern high-dimensional applications approximate nearest neighbor (ANN) search is typically used when a fast solution to the nearest neighbor problem is required.

Algorithms for ANN search can be divided into three categories: graphs (Malkov et al., 2014; Malkov and Yashunin, 2018; Iwasaki and Miyazaki, 2018; Baranchuk et al., 2019), quantization (Jegou et al., 2010; Johnson et al., 2019; Sablayrolles et al., 2019), and space-partitioning methods. In this article, we consider space-partitioning methods that can be further divided into tree-based (Muja and Lowe, 2014; Dasgupta and Sinha, 2015; Jääsaari et al., 2019) and hashing-based (Datar et al., 2004; Aumüller et al., 2019; Gong et al., 2020) algorithms that use trees and hash tables, respectively, as index structures.

Space-partitioning algorithms for ANN search use an index structure to select a *candidate set* $S(x) \subset \{1, \ldots, m\}$ of potential nearest neighbors. They then calculate the exact distances between the points in the candidate set and the query point, and return the k nearest points as the approximate nearest neighbors. These algorithms will correctly retrieve a nearest neighbor $j \in \text{NN}_k(x)$ if and only it belongs to the candidate set. Thus, the *recall* of a space-partitioning algorithm can be written as $\text{Rec}(S(x)) := \frac{1}{k}|\text{NN}_k(x) \cap S(x)|$, where we denote the number of elements of the set A by $|A|$. The performance of an approximate nearest neighbor algorithm is typically measured by its average *recall-query time tradeoff* (see e.g. Aumüller et al. (2019) or Li et al. (2019))—i.e., the average query time required to reach a certain average recall level on a set of test queries.

[1] In what follows, we assume that the ties are broken uniformly at random so that the query point always has exactly k nearest neighbors.

2.2 Multilabel classification

Consider a standard multi-label classification problem with m labels. Let $X \in \mathbb{R}^d$ be a random variable and let $L(X) \subseteq \{1, \dots, m\}$ be the corresponding label set. Equivalently, the output variable can be presented in binary encoding by defining $Y \in \{0,1\}^m$ as an m-bit random vector, where

$$Y_j = \begin{cases} 1, & \text{if } j \in L(X), \\ 0 & \text{otherwise.} \end{cases} \tag{2}$$

A multilabel classifier is an m-component function $g = (g_1, \dots, g_m) : \mathbb{R}^d \to \{0,1\}^m$ that attaches a label set to the value of the input variable X. Denote the training set that is assumed to be an i.i.d. sample from the distribution of the pair (X, Y) by $D_n := \{(X_i, Y_i)\}_{i=1}^n$. When the classifier $g : \mathbb{R}^d \times \{\mathbb{R}^d \times \{0,1\}^m\}^n \to \{0,1\}^m$ is learned from the training set of size n, we denote it by $g^{(n)}(x) := g(x, D_n)$. When the training set D_n is considered a random variable, the classifier $g^{(n)}$ also becomes a random function.

The performance of the classifier is measured by a loss function $L : \{0,1\}^m \times \{0,1\}^m \to \mathbb{R}$, and the objective is to minimize the risk $\mathcal{R}(g) := E[L(g(X), Y)]$. This risk is lower-bounded by the *Bayes risk* $\mathcal{R}^* = \inf_g \mathcal{R}(g)$, the minimizer of which is called the *Bayes classifier*.

The Bayes classifier for many common multilabel loss functions—such as Hamming loss, ranking loss, precision, recall, and F-measures—is obtained by thresholding the conditional label probabilities $\eta_j(x) := P\{Y_j = 1 \,|\, X = x\}$ (Dembczynski et al., 2010; Koyejo et al., 2015). This justifies the standard plug-in approach of first estimating the conditional label probabilities[2] by $\hat{\eta}_1(x), \dots, \hat{\eta}_m(x)$, and then defining the *plug-in classifier* as

$$g_j^{(n)}(x) := \begin{cases} 1, & \text{if } \hat{\eta}_j(x) > \tau \\ 0, & \text{otherwise,} \end{cases} \tag{3}$$

where $\tau \in [0,1]$; equivalently, the plug-in classifier can be written as the estimate of the label set $L(x)$ as $\hat{L}(x) := \{j \in \{1, \dots, m\} \,:\, \hat{\eta}_j(x) > \tau\}$.

The multilabel classification problem is often solved by reducing it to a series of binary or multiclass classification problems, and estimating the conditional label probabilities $\eta_j(x)$ under this model. (see, e.g., Menon et al. (2019) for a discussion of different reduction methods). In what follows, we will employ the *pick-all-labels* (PAL) reduction (Reddi et al., 2019) where we separate each label $l \in L(x_i)$ of the training set point x_i into a multi*class* (but single-label) training instance (x_i, l), and fit the classifier to this modified training set by minimizing a multiclass loss function.

3 Candidate set selection as a multilabel classification problem

Equipped with the above definitions, we are now in a position to formalize candidate set selection in ANN search described in Sec. 2.1 as an instance of the multilabel classification problem described in Sec. 2.2.

In the classical formulation of ANN search, the input–output pair is defined as $(x, \textsc{nn}_k(x))$. It is straightforward to observe that what we are dealing with is essentially an instance of the multilabel classification problem where $\textsc{nn}_k(x)$—i.e., the set of the indices of the k nearest neighbors of the query point—is the label set $L(x)$. Assuming that the values of x are i.i.d. draws from the distribution of the random variable X (the *query distribution*), the objective is to predict the value of the random variable Y (defined by (2) with $\textsc{nn}_k(X)$ as a label set) given the value of the random variable X. A distinctive property of this classification problem, which follows from the definition of the ANN task, is that the size of the label set $|L(x)| = \sum_{j=1}^m y_j = k$ is constant for all queries.

Since the labels $\{1, \dots, m\}$ correspond to the indices of the corpus points, the classification decision (3) where the probability estimates are thresholded corresponds to candidate set selection, and the estimated label set $\hat{L}(x)$ corresponds to the candidate set $S(x)$.

[2]More generally, instead of the conditional label probability estimates $\hat{\eta}_1(x), \dots, \hat{\eta}_m(x)$, any score function values $s_1(x), \dots, s_m(x)$ for the labels can be learned and thresholded to make the classification decision. While we will present all the results only for the version of the plug-in classifier that uses the probability estimates, they readily generalize to the version of the plug-in classifier that uses the score function values.

If no additional training data is available, the corpus itself can be used as a training set. More precisely, in this case we interpret $\{c_j\}_{j=1}^m$ as a sample from the query distribution, compute the k nearest neighbors of the corpus points, and then use $\{(c_j, y_j)\}_{j=1}^m$ as a training set. Note that in this case $y_{jj} = 1$ for each $j = 1, \ldots, m$ since each corpus point is the nearest neighbor of itself.

4 Partitioning classifiers

In this section, we first give a general definition of the partitioning classifier. Then we define the *natural classifier*—that is a special case of the partitioning classifier—for the single partition and for the ensemble of partitions for the general multilabel classification problem described in Sec. 2.2.

Partitioning classifier is a general term for a classifier that is based on learning a partition of the instance space and whose classification decision is based on the labels of the training set points that belong to the same partition element as the query point. Partitioning classifiers can be divided into two categories depending on whether the partition is flat or recursive. There is a vast literature on recursive partitioning classifiers (i.e., classification trees), and gradient-boosted trees (Friedman et al., 2000; Friedman, 2001) are one of the most widely used and efficient classifiers (Chen and Guestrin, 2016). Flat partitions are more typically used for density estimation (Kontkanen and Myllymäki, 2007; López-Rubio, 2013; Cui et al., 2021), but they have also been used for classification (Lugosi and Nobel, 1996; McAllester and Ortiz, 2003).

Denote by $\mathcal{P} = \{R_1, R_2, \ldots, R_L\}$ the partition of $\mathbb{R}^d$, i.e., a collection of disjoint sets for which $\bigcup_{l=1}^L R_l = \mathbb{R}^d$. Denote the structure function that maps the query point to the index of the partition element it belongs into by $q : \mathbb{R}^d \to \{1, 2, \ldots, L\}$. When the partition is learned from the training data, we denote it by $\mathcal{P}^{(n)} = \pi(D_n)$, where $\pi(D_n)$ is a partitioning rule that associates the training set with a partition of $\mathbb{R}^d$.

Natural classifier for a single partition. Partitioning classifiers use the training set twice: first, to learn the partition $\mathcal{P}^{(n)} = \pi(D_n)$, and second, to classify the query point using the training set points that belong to the same partition element $R_{q(x)}$ as the query point x. In the multiclass classification, the conditional label probabilities can be estimated in a natural fashion by the observed label proportions

$$\hat{\eta}_j(x) = \frac{1}{N_{q(x)}} \sum_{i\,:\,x_i \in R_{q(x)}} y_{ij}, \tag{4}$$

where $N_{q(x)} := |\{i \,:\, x_i \in R_{q(x)}\}|$ denotes the number of training set points in the same partition element as the query point. This standard practice can be motivated by noting that these observed label proportions are the maximum likelihood estimates of the piecewise constant multinomial model where the conditional label probabilities are constants at each of the partition elements.

In the multilabel case, the estimation of the conditional label probabilities by (4) can be motivated via the PAL reduction under which the observed label proportions in $R_{q(x)}$ are proportional to the maximum likelihood estimates of the piecewise constant multinomial model. To classify the query point, the conditional probability estimates (4) are plugged into (3), i.e., the query point is assigned into all the classes whose probability estimate is greater than or equal to the value of the threshold parameter τ. We call this partitioning classifier the *natural classifier*.

Natural classifier for an ensemble of partitions. When a collection of partitions $\{\mathcal{P}_t^{(n)}\}_{t=1}^T$, where $\mathcal{P}_t^{(n)} := \{R_1^{(t)}, \ldots, R_{L_t}^{(t)}\}$, is used as a classifier—such as in random forests (Ho, 1998; Breiman, 2001)—the contributions of the partitions are aggregated. In this article, we consider the most straightforward aggregation method where the conditional probability estimates are obtained as averages of the conditional probability estimates of the individual partitions:

$$\hat{\eta}_j(x) = \frac{1}{T} \sum_{t=1}^{T} \hat{\eta}_j^{(t)}(x). \tag{5}$$

The estimate of the single partition $\hat{\eta}_j^{(t)}(x)$ is defined as in (4) for the corresponding partition $\mathcal{P}_t^{(n)}$ and the corresponding structure function q_t.

5 Related work

The most directly relevant earlier literature consists of studies that learn space-partitioning index structures for ANN search using supervised information. The idea of optimising the index structure for the particular query distribution was first presented by Maneewongvatana and Mount (2001), and later extended by Cayton and Dasgupta (2008) who formulate ANN search as a supervised learning problem and propose a tree-based and a hashing-based algorithm for solving it. More recently, many supervised *learning to hash*-methods, such as minimal loss hashing (Norouzi and Fleet, 2011), LDA hashing (Strecha et al., 2011), and kernel-based supervised hashing (Liu et al., 2011), have also been proposed for ANN search (see, e.g., Wang et al. (2015) or Wang et al. (2017) for a survey).

However, the earlier supervised methods pose the supervised learning problem in an indirect fashion. This is because they, like the earlier unsupervised methods, select the candidate set using a method which we call *lookup search*[3]: they select the corpus point into the candidate set if and only if it belongs to the same partition element as the query point. Consequently, their objective is to learn a partition in which the k nearest neighbors of a query point belong to the same partition element with it. In contrast, our objective is to directly learn a partitioning classifier that predicts its nearest neighbors correctly. We will elucidate the difference in the next section.

6 Candidate set selection for ANN search

In view of the multilabel formulation of Sec. 3, the natural classifier defined in Sec. 4 can directly be used to select a candidate set for ANN search. However, in this section we interpret also the earlier lookup-based candidate set selection methods (lookup search and voting search) in our multilabel classification framework, and show that they define a classifier of a different form. We show that this classifier—that we call the *naive classifier*—is in fact the natural classifier for the different multilabel classification problem where the labels do not represent the corpus points that are the nearest neighbors of the query point but the corpus points that belong to the same partition element as the query point. This suggests that the natural classifier would be a more suitable candidate set selection method for ANN search than the naive classifier. The empirical results of Sec. 8 indicate that this is indeed the case.

Candidate set selection for a single partition. First, assume that we utilize the single fixed partition $\mathcal{P} = \{R_1, \ldots, R_L\}$ and the training set $\{(x_i, y_i)\}_{i=1}^n$ to approximate the nearest neighbors of the query point x. The natural classifier defined in Sec. 4 selects the candidate set as

$$\hat{L}(x) = \{j \in \{1, \ldots, m\} \,|\, \hat{\eta}_j(x) > \tau\}, \tag{6}$$

where $\tau \in [0, 1]$, and the conditional label probability estimates $\hat{\eta}_j(x) = \frac{1}{N_{q(x)}} \sum_{i\,:\,x_i \in R_{q(x)}} y_{ij}$ are obtained as the observed label proportions among the training set points that belong to the same partition element with the query point. In contrast, lookup search selects the candidate set as

$$\hat{L}(x) = \{j \in \{1, \ldots, m\} \,|\, c_j \in R_{q(x)}\}, \tag{7}$$

i.e., it selects the corpus point into the candidate set if and only if it belongs to the same partition element with the query point. When interpreted in the classification framework of Sec. 3, (7) defines the classifier $\hat{L}(x) = \{j \in \{1, \ldots, m\} \,|\, \tilde{\eta}_j(x) > \tau\}$, where $\tau \in [0, 1)$ and $\tilde{\eta}_j(x) = \mathbb{1}_{R_{q(x)}}(c_j)$; we call this a *naive classifier*.

We immediately observe that the naive classifier is not a natural classifier for the multilabel classification problem in which the labels are defined as $y_{ij} = \mathbb{1}_{\mathrm{NN}_k(x_i)}(c_j)$ as in Sec. 3. Instead, it is a natural classifier for the different multilabel classification problem in which the labels are defined

[3] Often, lookup search is combined with a priority queue guided backtracking search in which the query point is routed into more than one element in a partition, and all the corpus points that belong to these partition elements are then selected into the candidate set. This technique is called *priority search* (Arya and Mount, 1993; Silpa-Anan and Hartley, 2008) or *multi-probe LSH* (Lv et al., 2007), depending on whether the index structure is tree-based or hashing-based, respectively. For clarity, we do not consider backtracking—that is mainly a memory-saving technique—in the analysis below: it can be easily incorporated both into lookup search and our method by allowing the structure function q to return a set of indices instead of a single index, and considering $\bigcup_{l \in q(x)} R_l$ instead of the single partition element $R_{q(x)}$.

as $\tilde{y}_{ij} = \mathbb{1}_{R_q(x_i)}(c_j)$. In other words, the naive classifier is geared towards the learning problem in which—instead of the k nearest neighbors of the query point—the labels represent the corpus points that belong to the same partition element as the query point. The candidate set selection method (7) also explains why the objective of the earlier supervised methods for *learning* the partition differs from ours: in these methods, the objective is to maximise the number of nearest neighbors of the query point that belong to the same partition element with it in order to maximise the recall (while minimising the number of non-neighbors in that element in order to maximise precision).

Candidate set selection for an ensemble of partitions. Assume that the fixed set of partitions $\{\mathcal{P}_t\}_{t=1}^T$ is used to approximate the k nearest neighbors of a query point x. The natural classifier defined in Sec. 4 selects the candidate set

$$\hat{L}(x) = \{j \in \{1, \ldots, m\} \mid \hat{\eta}_j(x) > \tau\}, \tag{8}$$

as in (6), but now $\hat{\eta}_j(x) = \frac{1}{T}\sum_{t=1}^T \hat{\eta}_j^{(t)}(x)$, where the contributions of the individual partitions $\hat{\eta}_j^{(t)}(x)$ are defined as above. In contrast, the earlier (both supervised and unsupervised) methods select the corpus point into the candidate set if and only if it belongs to the same partition element as the query point in at least one of the T partitions. Hence, the candidate set selected by lookup search is

$$\hat{L}(x) = \left\{ j \in \{1, \ldots, m\} \mid c_j \in \bigcup_{t=1}^T R_{q^{(t)}(x)}^{(t)} \right\} = \{j \in \{1, \ldots, m\} \mid \tilde{\eta}_j(x) > \tau\}, \tag{9}$$

where $\tilde{\eta}_j(x) := \frac{1}{T}\sum_{t=1}^T \tilde{\eta}_j^{(t)}(x)$, $\tau \in [0, \frac{1}{T})$, and the contributions of the partitions $\tilde{\eta}_j^{(t)}(x)$ are defined as above.

Unlike in the case of a single partition—where the value of the threshold parameter $\tau \in [0, 1)$ does not affect the classification decision of the naive classifier, since $\tilde{\eta}_j(x) \in \{0, 1\}$—now τ affects the classification decision, since $\tilde{\eta}_j(x) \in \{0, \frac{1}{T}, \ldots, \frac{T-1}{T}, 1\}$. Hence, a tuning parameter can be added to lookup search by allowing τ to be chosen freely as proposed by Hyvönen et al. (2016) who call the resulting method *voting search*.

7 Consistency of partitioning classifiers for ANN search

The ideal index structure for ANN search always returns a candidate set that contains all the k nearest neighbors of the query point and no other corpus points. Under the multilabel formulation, this corresponds to a classifier for which the expected multilabel 0-1 loss $EL(g(X), Y) = P\{g(X) \neq Y\}$ is zero. To this end, we prove a sufficient condition for the consistency of a partitioning classifier for ANN search under 0-1 loss. Consistency under 0-1 loss also directly implies consistency for the other common multilabel loss functions, such as Hamming loss, precision, recall, and F-measures. As a concrete example, we prove the consistency of the chronological k-d tree (Bentley, 1975) by checking that this condition holds for it.

7.1 Sufficient condition for consistency

The classical theorem for proving consistency of partitioning classifiers for binary classification is:

Theorem 1. *(Devroye et al. (1996), Theorem 6.1, p. 94–95) Assume that only the features $X_1, \ldots, X_n$ are used to learn the partition $\mathcal{P}^{(n)} = \pi(X_1, \ldots, X_n)$. The natural classifier[4] $g^{(n)}$ defined by $\mathcal{P}^{(n)}$ is consistent (under 0-1 loss) for binary classification, if*

(i) $N_{q(X)} \to \infty$ *in probability, and*

(ii) $\operatorname{diam}\left(R_{q(X)}\right) \to 0$ *in probability,*

when $n \to \infty$.

[4]The natural classifier for binary classification is defined as the classifier that classifies the query point into the majority class of the training set points that belong to the same partition element with it.

The number of the training set points in the partition element the query point x belongs to is denoted by $N_q(x) := |\{i : X_i \in R_{q(x)}\}|$, and the diameter of a set A is defined as the maximum distance between any two points of this set and denoted by $\mathrm{diam}(A) := \sup_{a,\, b \in A} \|a - b\|$.

While this result is for binary classification, it can be readily extended to the multilabel case. However, as a multilabel classification problem ANN search has two distinguishing properties: (i) the Bayes error $\mathcal{R}^*$ is zero; (ii) decision boundaries between the labels consist of subsets of hyperplanes. It turns out that in this case, the second condition in Theorem 1 is sufficient for the consistency of a partitioning classifier[5]:

Theorem 2. *Let $g^{(n)}$ be a natural classifier defined by the partition $\mathcal{P}^{(n)} = (R_1, \ldots, R_L)$ and the threshold parameter $\tau \in [0, 1)$ for ANN search. Assume that the distribution of X, denoted by μ, is continuous. If $\mathrm{diam}(R_{q(X)}) \to 0$ in probability—that is, if for every $\epsilon > 0$,*

$$P\{\mathrm{diam}(R_{q(X)}) > \epsilon\} \to 0$$

when $n \to \infty$, then the classifier $g^{(n)}$ is consistent (for 0-1 loss)—i.e., $E_{D_n}\mathcal{R}(g^{(n)}) \to 0$.

Proof. If for all the pairs of corpus points $(c_j, c_{j'})$, $j' \neq j$, all the points of the partition element R_l are closer to c_j than $c_{j'}$ (or vice versa)—that is, if there is no such pair $(c_j, c_{j'})$ for which there exists $a, b \in R_l$ such that $\|a - c_j\| < \|a - c_{j'}\|$ and $\|b - c_j\| > \|b - c_{j'}\|$—then also $\hat{\eta}_j(x) = \eta_j(x)$ for each $x \in R_l$ and $j = 1, \ldots, m$; consequently, each $x \in R_l$ is classified correctly for any $\tau \in [0, 1)$. Now, since for each $j = 1, \ldots, m$,

$$\begin{aligned}
&P\{g_j^{(n)}(X) \neq \eta_j(X)\} \\
&\leq P\left(\exists j' \neq j : \exists a, b \in R_{q(X)} \text{ s.t. } \|a - c_j\| < \|a - c_{j'}\|,\, \|b - c_j\| > \|b - c_{j'}\|\right) \\
&\leq \sum_{j' \neq j} P\{\exists a, b \in R_{q(X)} \text{ s.t. } \|a - c_j\| < \|a - c_{j'}\|,\, \|b - c_j\| > \|b - c_{j'}\|\},
\end{aligned}$$

to prove consistency of $g^{(n)}$ it is sufficient to show that for all $j, j' \in \{1, \ldots, m\}$, $j \neq j'$,

$$P\{\exists a, b \in R_{q(X)} \text{ s.t. } \|a - c_j\| < \|a - c_{j'}\|,\, \|b - c_j\| > \|b - c_{j'}\|\} \to 0$$

in probability when $n \to \infty$.

Choose any j, j', $j \neq j'$, and denote the hyperplane that is halfway in between the corpus points c_j and $c_{j'}$ by $H := \{x \in \mathbb{R}^d : \|x - c_j\| = \|x - c_{j'}\|\}$. For any $t = 1, 2, \ldots$, let H_t denote the set surrounding H by a margin of width $1/t$. Since $H_1 \supset H_2 \supset H_3 \ldots$, and $H = \cap_{t=1}^{\infty} H_t$, it follows from the upper continuity of the probability measure that $\lim_{t \to \infty} \mu(H_t) = \mu(H)$. Because the Lebesgue measure of the hyperplane H in $\mathbb{R}^d$ is zero and μ is absolutely continuous w.r.t. the Lebesgue measure by the assumption, then also $\lim_{t \to \infty} \mu(H_t) = \mu(H) = 0$.

Now, for any $t = 1, 2, \ldots$, if $R_{q(x)}$ crosses the hyperplane H, then either $x \in H_t$ or the diameter of the $R_{q(x)}$ is greater than $1/t$. Hence,

$$\begin{aligned}
&P\{\exists a, b \in R_{q(X)} \text{ s.t. } \|a - c_j\| < \|a - c_{j'}\|,\, \|b - c_j\| > \|b - c_{j'}\|\} \\
&\leq P\{X \in H_t \text{ or } \mathrm{diam}(R_{q(X)}) > 1/t\} \\
&\leq \mu(H_t) + P\{\mathrm{diam}(R_{q(X)}) > 1/t\}.
\end{aligned}$$

We can get $\mu(H_t)$ as small as desired by choosing a large enough t; and since by assumption the second term is arbitrarily small when n is large enough, the result follows. □

7.2 Consistency of chronological k-d tree

Next, we illustrate the utility of Theorem 2 by applying it to prove the consistency of the *chronological k-d tree* (Bentley, 1975) that rotates the split directions and uses the same split direction for all the nodes at one level of a tree. At the first level the training data is split at the median of the first coordinates of the data points. At the second level both nodes are split at the median of the second

[5]The proofs of the results of this section are presented in Appendix A.

coordinates of the node points. At the $(d+1)$th level, the nodes are split again at the median of the first coordinates, and so on (see Appendix C.1).

More precisely, let $X, X_1, \ldots, X_n \in \mathbb{R}^d$ be i.i.d. random variables. A chronological k-d tree can be formalized as a partitioning rule π that returns the partition $\mathcal{P}^{(n)} = \pi(X_1, \ldots, X_n)$. When the tree height is ℓ, this partition has 2^ℓ elements (also called *leafs*). The leafs are hyperrectangles in $\mathbb{R}^d$. Some of the edges of these hyperrectangles may have an infinite length. To handle these leafs, we introduce the notation where, for any $M > 0$, the hypercube $[-M, M]^d$ divides the partition elements $R_1, \ldots, R_{2^\ell}$ into three disjoint sets:

$$\begin{aligned} A &:= \{l \in \{1, \ldots, 2^\ell\} \,:\, R_l \subset [-M, M]^d\}, \\ C &:= \{l \in \{1, \ldots, 2^\ell\} \,:\, R_l \subset \mathbb{R}^d \setminus [-M, M]^d\}, \\ B &:= \{1, \ldots, 2^\ell\} \setminus (A \cup C). \end{aligned} \tag{10}$$

Here A is the set of indexes of the partition elements that are completely inside the hypercube $[-M, M]^d$, B is the set of indexes of the partition elements that cross its boundary, and C is the set of indexes of the partition elements that are completely outside of it.

First, we prove two auxiliary results that bound the number of nodes crossing the boundary of the box $[-M, M]^d$ and the combined length of the edges (in any fixed coordinate direction) of the nodes that reside completely inside $[-M, M]^d$, respectively. Note that these bounds are of purely combinatorial nature and thus do not depend on the training set.

Lemma 1. *For any training set D_n, it holds for the number of nodes of a chronological k-d tree—denoted by $N_B := |B|$—crossing the border of the hypercube $[-M, M]^d$ that*

$$N_B \leq 4d \cdot 2^{\ell - \frac{\ell}{d}}.$$

Lemma 2. *Let $j \in \{1, \ldots, d\}$ be any coordinate direction. Denote the length of the node R_l in the jth coordinate direction by V_l. Then for any training set D_n,*

$$\sum_{l \in A} V_l \leq 4M \cdot 2^{\ell - \frac{\ell}{d}}.$$

We are now in a position to establish the consistency of the chronological k-d tree for approximate nearest neighbor search. In view of Theorem 2 it suffices to prove that the leaf diameter converges to zero in probability:

Theorem 3. *If for the height of a chronological k-d tree holds that $\ell \to \infty$ when $n \to \infty$, then the leaf diameter $\operatorname{diam}(R_{q(X)})$ converges to zero in probability.*

8 Experiments

We present empirical results validating the utility of our framework. In particular, we compare the natural classifier to the earlier candidate set selection methods discussed in Sec. 6 for different types of unsupervised trees that have been widely used for ANN search. Specifically, we use ensembles of randomized k-d trees (Friedman et al., 1976; Silpa-Anan and Hartley, 2008), random projection (RP) trees (Dasgupta and Freund, 2008; Hyvönen et al., 2016), and principal component (PCA) trees (Sproull, 1991; Jääsaari et al., 2019) (see Appendix C for detailed descriptions of these data structures). Another consequence of the multilabel formulation of Sec. 3 is that it enables using any established multilabel classifier for ANN search. To demonstrate this concretely, we train a random forest consisting of standard multilabel classification trees (trained under the PAL reduction (Reddi et al., 2019) by using multinomial log-likelihood as a split criterion) and use it as an index structure for ANN search; it turns out that the fully supervised classification trees have an improved performance compared to the earlier unsupervised trees on some—but, curiously, not on all—data sets.

We follow a standard ANN search performance evaluation setting (Aumüller et al., 2019; Li et al., 2019) by using the corpus as the training set, searching for $k = 10$ nearest neighbors in Euclidean distance, and measuring performance by evaluating average recall and query time over the test set of 1000 points. We use four benchmark data sets: Fashion ($m = 60000$, $d = 784$), GIST ($m = 1000000$, $d = 960$), Trevi ($m = 101120$, $d = 4096$), and STL-10 ($m = 100000$, $d = 9216$). All the algorithms

are implemented in C++ and ran using a single thread. We tune the hyperparameters by grid search and plot the Pareto frontiers of the optimal hyperparameters. Further details of the experimental setup are found in Appendix B.

Comparison of candidate set selection methods. The candidate set selection method proposed in this article is the natural classifier (8) described in Sec. 4; for completeness, we also include the special case obtained by fixing $\tau = 0$ in the comparison. The earlier methods are lookup search (naive classifier (9) with $\tau = 0$) and voting (Hyvönen et al., 2016; Jääsaari et al., 2019) (naive classifier (9) with τ as a free tuning parameter). The results for the Trevi data set are presented in Fig. 1 and indicate, as the discussion of Sec. 6 suggests, that the natural classifier performs better than the earlier lookup-based methods for all types of trees (this finding holds consistently over all the data sets in our experiments; see Fig. 2 in Appendix).

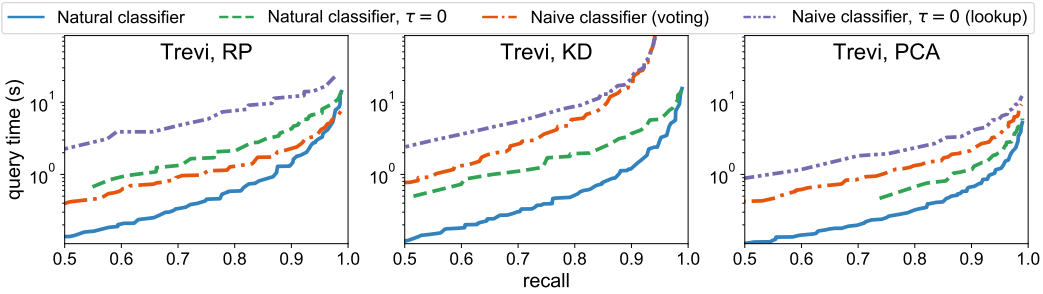

Figure 1: Recall vs. query time (log scale) of ensembles of, RP, k-d, and PCA trees. The solid blue line is the natural classifier proposed in this paper; the dash-dotted red line is the natural classifier with $\tau = 0$ that is included for completeness; the dashed green line is voting; and the double-dash-dotted violet line is lookup search. The natural classifier is the fastest and the lookup search is the slowest of the methods for each tree type.

Comparison of tree types. We compare the aforementioned ensembles of unsupervised (RP, KD, and PCA) trees and the random forest consisting of supervised classification trees (RF); for all four tree types the candidate set is selected by (8). The results are shown in Table 1. We expected that the random forest (RF) would be the fastest tree-based method, since it leverages supervised information to learn the trees. Indeed, this is the case on Fashion and GIST. However, on STL-10 and Trevi, the unsupervised PCA tree is the fastest method. We hypothesize that this is because of the high dimensionality of STL-10 and Trevi: standard supervised classification trees employed by random forest are restricted to axis-aligned splits, whereas PCA trees—although they use an unsupervised split criterion—can find more informative oblique split directions. An interesting topic for future work would be to apply supervised classification trees that can utilize oblique split directions.

9 Conclusion

We establish a general theoretical framework for ANN search by formulating candidate set selection as a multilabel learning task. Empirical results validate our framework: a natural classifier derived directly from the problem formulation is a strict improvement over the earlier lookup-based candidate set selection methods. In addition, we provide a sufficient condition that guarantees consistency of a partitioning classifier for ANN search. We verify this condition for chronological k-d trees, indicating that—given enough training data—they retrieve a candidate set containing all the k nearest neighbors of the query point and no other corpus points.

Limitations. Supervised ANN search methods typically have longer pre-processing times compared to unsupervised methods. This is because (1) they require computing the true nearest neighbors $\{y_i\}_{i=1}^n$ of the training set points $\{x_i\}_{i=1}^n$ and (2) supervised index structures are often slower to build compared to their unsupervised counterparts (c.f. Appendix D.1). If fast index construction is required, the second problem can be mitigated by learning trees in an unsupervised fashion, but using them as partitioning classifiers as described in Sec. 4, since the experiments of Sec. 8 suggest that the

Table 1: Query times (seconds / 1000 queries) at different recall levels for the different tree types. The fastest method in each case is typeset in boldface.

data set	R (%)	PCA	KD	RP	RF
Fashion	80	0.075	0.076	0.099	**0.063**
	90	0.111	0.126	0.172	**0.095**
	95	0.163	0.171	0.261	**0.146**
GIST	80	1.330	0.958	1.009	**0.705**
	90	2.942	2.286	2.226	**1.530**
	95	5.641	4.451	4.598	**3.253**
STL-10	80	**0.382**	0.872	1.211	0.756
	90	**0.756**	2.126	3.248	1.774
	95	**1.315**	4.376	7.330	3.654
Trevi	80	**0.330**	0.543	0.591	0.582
	90	**0.684**	1.464	1.468	1.234
	95	**1.212**	3.244	3.289	2.350

candidate set selection method has a more pronounced effect on the performance than than the tree type.

Future research directions. While we demonstrate our approach using a random forest classifier, we expect that the most important consequence of our work is that it enables using any type of classifier as an index structure for ANN search. In particular, gradient-boosted trees (Friedman, 2001) are promising, since they are generally more accurate than random forests. *Extreme classification* models, including tree-based models (Agrawal et al., 2013; Prabhu and Varma, 2014; Jain et al., 2016), sparse linear models (Babbar and Schölkopf, 2017, 2019; Yen et al., 2017), and embedding-based neural networks (Guo et al., 2019), are also promising model candidates for ANN search since they are specifically tailored for multilabel classification problems with extremely large label spaces.

Our formulation enables analyzing ANN search in the statistical learning framework, thus opening multiple theoretical research questions: (1) Can we establish a sufficient condition for *strong* consistency? (2) Can we prove consistency of more adaptive partitioning classifiers, such as PCA trees or classification trees? (3) Can we establish faster than logarithmic convergence rates? The last question is especially interesting, since prediction times of trees are logarithmic: a positive answer would theoretically justify decreasing query times by increasing the training set size.

Acknowledgements

Funding in direct support of this work: Academy of Finland grants #345635 (DAISY), #311277 (TensorML), and #313857 (WiFiUS). The authors wish to thank the Finnish Computing Competence Infrastructure (FCCI) for supporting this project with computational and data storage resources.

References

Rahul Agrawal, Archit Gupta, Yashoteja Prabhu, and Manik Varma. Multi-label learning with millions of labels: Recommending advertiser bid phrases for web pages. In *Proceedings of the 22nd International Conference on World Wide Web*, pages 13–24. ACM, 2013.

Sunil Arya and David M Mount. Algorithms for fast vector quantization. In *[Proceedings] DCC93: Data Compression Conference*, pages 381–390. IEEE, 1993.

Martin Aumüller, Erik Bernhardsson, and Alexander Faithfull. ANN-benchmarks: A benchmarking tool for approximate nearest neighbor algorithms. *Information Systems*, 87, 2019.

Martin Aumüller, Tobias Christiani, Rasmus Pagh, and Michael Vesterli. PUFFINN: parameterless and universally fast finding of nearest neighbors. In *27th Annual European Symposium on Algorithms, ESA 2019*, pages 10:1–10:16, 2019.

Rohit Babbar and Bernhard Schölkopf. DiSMEC: Distributed sparse machines for extreme multi-label classification. In *Proceedings of the Tenth ACM International Conference on Web Search and Data Mining*, pages 721–729, 2017.

Rohit Babbar and Bernhard Schölkopf. Data scarcity, robustness and extreme multi-label classification. *Machine Learning*, 108(8-9):1329–1351, 2019.

Dmitry Baranchuk, Dmitry Persiyanov, Anton Sinitsin, and Artem Babenko. Learning to route in similarity graphs. In *International Conference on Machine Learning*, pages 475–484. PMLR, 2019.

Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.

Leo Breiman. Random forests. *Machine Learning*, 45(1):5–32, 2001.

Lawrence Cayton and Sanjoy Dasgupta. A learning framework for nearest neighbor search. In *Advances in Neural Information Processing Systems*, pages 233–240, 2008.

Tianqi Chen and Carlos Guestrin. XGBoost: A scalable tree boosting system. In *Proceedings of the 22nd ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 785–794. ACM, 2016.

Jingyi Cui, Hanyuan Hang, Yisen Wang, and Zhouchen Lin. GBHT: Gradient boosting histogram transform for density estimation. In *International Conference on Machine Learning*, pages 2233–2243. PMLR, 2021.

Sanjoy Dasgupta and Yoav Freund. Random projection trees and low dimensional manifolds. In *Proceedings of the 40th Annual ACM Symposium on Theory of Computing*, pages 537–546, 2008.

Sanjoy Dasgupta and Kaushik Sinha. Randomized partition trees for nearest neighbor search. *Algorithmica*, 72(1):237–263, 2015.

Mayur Datar, Nicole Immorlica, Piotr Indyk, and Vahab S Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *Proceedings of the 20th Annual Symposium on Computational Geometry*, pages 253–262, 2004.

Krzysztof Dembczynski, Weiwei Cheng, and Eyke Hüllermeier. Bayes optimal multilabel classification via probabilistic classifier chains. In *International Conference on Machine Learning*. PMLR, 2010.

Luc Devroye, László Györfi, and Gábor Lugosi. *A Probabilistic Theory of Pattern Recognition*. Springer Science & Business Media, 1996.

Jerome Friedman, Trevor Hastie, Robert Tibshirani, et al. Additive logistic regression: a statistical view of boosting (with discussion and a rejoinder by the authors). *The Annals of Statistics*, 28(2): 337–407, 2000.

Jerome H Friedman. Greedy function approximation: a gradient boosting machine. *Annals of Statistics*, pages 1189–1232, 2001.

Jerome H Friedman, Jon Louis Bentley, and Raphael Ari Finkel. An algorithm for finding best matches in logarithmic time. *ACM Trans. Math. Software*, 3(SLAC-PUB-1549-REV. 2):209–226, 1976.

Long Gong, Huayi Wang, Mitsunori Ogihara, and Jun Xu. iDEC: indexable distance estimating codes for approximate nearest neighbor search. *Proceedings of the VLDB Endowment*, 13(9), 2020.

Michael Greenwald and Sanjeev Khanna. Space-efficient online computation of quantile summaries. *ACM SIGMOD Record*, 30(2):58–66, 2001.

Chuan Guo, Ali Mousavi, Xiang Wu, Daniel N Holtmann-Rice, Satyen Kale, Sashank Reddi, and Sanjiv Kumar. Breaking the glass ceiling for embedding-based classifiers for large output spaces. In *Advances in Neural Information Processing Systems*, pages 4944–4954, 2019.

Ruiqi Guo, Philip Sun, Erik Lindgren, Quan Geng, David Simcha, Felix Chern, and Sanjiv Kumar. Accelerating large-scale inference with anisotropic vector quantization. In *International Conference on Machine Learning*, pages 3887–3896. PMLR, 2020.

Tin Kam Ho. The random subspace method for constructing decision forests. *IEEE transactions on pattern analysis and machine intelligence*, 20(8):832–844, 1998.

Ville Hyvönen, Teemu Pitkänen, Sotiris Tasoulis, Elias Jääsaari, Risto Tuomainen, Liang Wang, Jukka Corander, and Teemu Roos. Fast nearest neighbor search through sparse random projections and voting. In *Proceedings of the 4th IEEE International Conference on Big Data*, pages 881–888. IEEE, 2016.

Piotr Indyk and Rajeev Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proceedings of the 30th Annual ACM Symposium on Theory of Computing*, pages 604–613, 1998.

Masajiro Iwasaki and Daisuke Miyazaki. Optimization of indexing based on k-nearest neighbor graph for proximity search in high-dimensional data. *arXiv preprint arXiv:1810.07355*, 2018.

Elias Jääsaari, Ville Hyvönen, and Teemu Roos. Efficient autotuning of hyperparameters in approximate nearest neighbor search. In *Proceedings of the 23rd Pacific-Asia Conference on Knowledge Discovery and Data Mining*, volume 2, pages 590–602. Springer, 2019.

Himanshu Jain, Yashoteja Prabhu, and Manik Varma. Extreme multi-label loss functions for recommendation, tagging, ranking & other missing label applications. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 935–944, 2016.

Herve Jegou, Matthijs Douze, and Cordelia Schmid. Product quantization for nearest neighbor search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 33(1):117–128, 2010.

Jeff Johnson, Matthijs Douze, and Hervé Jégou. Billion-scale similarity search with GPUs. *IEEE Transactions on Big Data*, 2019.

Petri Kontkanen and Petri Myllymäki. MDL histogram density estimation. In *Artificial Intelligence and Statistics*, pages 219–226. PMLR, 2007.

Oluwasanmi O Koyejo, Nagarajan Natarajan, Pradeep K Ravikumar, and Inderjit S Dhillon. Consistent multilabel classification. *Advances in Neural Information Processing Systems*, 28:3321–3329, 2015.

Der-Tsai Lee and Chak-Kuen Wong. Worst-case analysis for region and partial region searches in multidimensional binary search trees and balanced quad trees. *Acta Informatica*, 9(1):23–29, 1977.

Wen Li, Ying Zhang, Yifang Sun, Wei Wang, Mingjie Li, Wenjie Zhang, and Xuemin Lin. Approximate nearest neighbor search on high dimensional data-experiments, analyses, and improvement. *IEEE Transactions on Knowledge and Data Engineering*, pages 1475–1488, 2019.

Wei Liu, Jun Wang, Sanjiv Kumar, and Shih-Fu Chang. Hashing with graphs. In *Proceedings of the 28th International Conference on Machine Learning*, pages 1–8, 2011.

Ezequiel López-Rubio. A histogram transform for probability density function estimation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 36(4):644–656, 2013.

Gábor Lugosi and Andrew Nobel. Consistency of data-driven histogram methods for density estimation and classification. *The Annals of Statistics*, 24(2):687–706, 1996.

Qin Lv, William Josephson, Zhe Wang, Moses Charikar, and Kai Li. Multi-probe LSH: efficient indexing for high-dimensional similarity search. In *Proceedings of the 33rd International Conference on Very Large Data Bases*, pages 950–961, 2007.

Yury Malkov, Alexander Ponomarenko, Andrey Logvinov, and Vladimir Krylov. Approximate nearest neighbor algorithm based on navigable small world graphs. *Information Systems*, 45:61–68, 2014.

Yury A Malkov and Dmitry A Yashunin. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2018.

Songrit Maneewongvatana and David M Mount. The analysis of a probabilistic approach to nearest neighbor searching. In *Workshop on Algorithms and Data Structures*, pages 276–286. Springer, 2001.

David McAllester and Luis Ortiz. Concentration inequalities for the missing mass and for histogram rule error. *Journal of Machine Learning Research*, 4(Oct):895–911, 2003.

Mark McCartin-Lim, Andrew McGregor, and Rui Wang. Approximate principal direction trees. In *Proceedings of the 29th International Coference on International Conference on Machine Learning*, pages 1611–1618, 2012.

Aditya K Menon, Ankit Singh Rawat, Sashank Reddi, and Sanjiv Kumar. Multilabel reductions: what is my loss optimising? *Advances in Neural Information Processing Systems*, 32, 2019.

Marius Muja and David G Lowe. Scalable nearest neighbor algorithms for high dimensional data. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 36(11):2227–2240, 2014.

Mohammad Norouzi and David J Fleet. Minimal loss hashing for compact binary codes. In *28th International Conference on on Machine Learning*, pages 353–360, 2011.

Yashoteja Prabhu and Manik Varma. FastXML: A fast, accurate and stable tree-classifier for extreme multi-label learning. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 263–272, 2014.

Sashank J Reddi, Satyen Kale, Felix Yu, Daniel Holtmann-Rice, Jiecao Chen, and Sanjiv Kumar. Stochastic negative mining for learning with large output spaces. In *The 22nd International Conference on Artificial Intelligence and Statistics*, pages 1940–1949. PMLR, 2019.

Alexandre Sablayrolles, Matthijs Douze, Cordelia Schmid, and Hervé Jégou. Spreading vectors for similarity search. In *Proceedings of the 7th International Conference on Learning Representations*, 2019.

Chanop Silpa-Anan and Richard Hartley. Optimised kd-trees for fast image descriptor matching. In *2008 IEEE Conference on Computer Vision and Pattern Recognition*, pages 1–8. IEEE, 2008.

Robert F Sproull. Refinements to nearest-neighbor searching in k-dimensional trees. *Algorithmica*, 6 (1-6):579–589, 1991.

Christoph Strecha, Alex Bronstein, Michael Bronstein, and Pascal Fua. LDAHash: Improved matching with smaller descriptors. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 34 (1):66–78, 2011.

Vladimir Naumovich Vapnik and Alexey Yakovlevich Chervonenkis. On the uniform convergence of relative frequencies of events to their probabilities. *Theory of Probability & Its Applications*, 16 (2):264–280, 1971.

Jingdong Wang, Ting Zhang, Nicu Sebe, Heng Tao Shen, et al. A survey on learning to hash. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 40(4):769–790, 2017.

Jun Wang, Wei Liu, Sanjiv Kumar, and Shih-Fu Chang. Learning to hash for indexing big data – a survey. *Proceedings of the IEEE*, 104(1):34–57, 2015.

Yair Weiss, Antonio Torralba, and Rob Fergus. Spectral hashing. In *Advances in Neural Information Processing Systems*, pages 1753–1760, 2009.

Ian EH Yen, Xiangru Huang, Wei Dai, Pradeep Ravikumar, Inderjit Dhillon, and Eric Xing. PPDSparse: A parallel primal-dual sparse method for extreme classification. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 545–553, 2017.

Qi Zhang and Wei Wang. A fast algorithm for approximate quantiles in high speed data streams. In *19th International Conference on Scientific and Statistical Database Management (SSDBM 2007)*, pages 29–29. IEEE, 2007.

Checklist

1. For all authors...
   (a) Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope? [Yes]
   (b) Did you describe the limitations of your work? [Yes] See Section 9
   (c) Did you discuss any potential negative societal impacts of your work? [N/A]
   (d) Have you read the ethics review guidelines and ensured that your paper conforms to them? [Yes]
2. If you are including theoretical results...
   (a) Did you state the full set of assumptions of all theoretical results? [Yes]
   (b) Did you include complete proofs of all theoretical results? [Yes] See Appendix A
3. If you ran experiments...
   (a) Did you include the code, data, and instructions needed to reproduce the main experimental results (either in the supplemental material or as a URL)? [Yes] See the supplemental repository
   (b) Did you specify all the training details (e.g., data splits, hyperparameters, how they were chosen)? [Yes] See Appendix B and the supplemental repository
   (c) Did you report error bars (e.g., with respect to the random seed after running experiments multiple times)? [No]
   (d) Did you include the total amount of compute and the type of resources used (e.g., type of GPUs, internal cluster, or cloud provider)? [No]
4. If you are using existing assets (e.g., code, data, models) or curating/releasing new assets...
   (a) If your work uses existing assets, did you cite the creators? [N/A]
   (b) Did you mention the license of the assets? [N/A]
   (c) Did you include any new assets either in the supplemental material or as a URL? [N/A]
   (d) Did you discuss whether and how consent was obtained from people whose data you're using/curating? [N/A]
   (e) Did you discuss whether the data you are using/curating contains personally identifiable information or offensive content? [N/A]
5. If you used crowdsourcing or conducted research with human subjects...
   (a) Did you include the full text of instructions given to participants and screenshots, if applicable? [N/A]
   (b) Did you describe any potential participant risks, with links to Institutional Review Board (IRB) approvals, if applicable? [N/A]
   (c) Did you include the estimated hourly wage paid to participants and the total amount spent on participant compensation? [N/A]

A Proofs

Proof of Lemma 1

Proof. The border of the hypercube consists of $(d-1)$-dimensional faces. Choose a coordinate direction $j \in \{1, \ldots, d\}$ and consider a $(d-1)$-face that is orthogonal to that coordinate axis. Denote the number of nodes crossing that $(d-1)$-face by $N_B^{(j)}$. Before any splits there is one node—the whole feature space $\mathbb{R}^d$—that crosses it. Splitting a node that crosses that $(d-1)$-face at a coordinate direction other than j creates two nodes crossing it if the splitting hyperplane intersects with $[-M, M]^d$ (if the splitting hyperplane does not intersect with $[-M, M]^d$ then it does affect $N_B^{(j)}$). Splitting at the jth direction does not increase $N_B^{(j)}$ since the splitting hyperplane is perpendicular to the $(d-1)$-face we consider and so cannot cross it.

Therefore, if ℓ is a multiple of d, we have $N_B^{(j)} \leq 2^{\ell - \frac{\ell}{d}}$ since each full round of d splits contains $d-1$ splits orthogonal to the jth coordinate direction, each of which may double the number of nodes crossing the $(d-1)$-face, and one split parallel to the jth coordinate direction that doesn't increase the number. If ℓ is not a multiple of d, then

$$N_B^{(j)} \leq 2^{\ell - \lfloor \frac{\ell}{d} \rfloor} \leq 2^{\ell - \frac{\ell}{d} + 1}$$

because the last incomplete round of splits may not contain a split at the jth coordinate direction. Since for each coordinate direction a d-dimensional hypercube has two $(d-1)$-faces that are orthogonal to that coordinate axis, we have[6]

$$N_B \leq 2 \sum_{j=1}^{d} N_B^{(j)} \leq 4d \cdot 2^{\ell - \frac{\ell}{d}}.$$

□

Proof of Lemma 2

For each $l = 1, \ldots, 2^\ell$, denote the length of the hyperrectangle $R_l' := R_l \cap [-M, M]^d$ in the jth coordinate direction by V_l'. Clearly, $V_l = V_l'$ for each $l \in A$ by the definition of the set A. Thus,

$$\sum_{l \in A} V_l = \sum_{l \in A} V_l' \leq \sum_{l=1}^{2^\ell} V_l'.$$

Before any splits, there is one node with $V_l' = 2M$. Splitting a node in a coordinate direction other than j creates two child nodes with the same length in the jth coordinate direction as the parent node, and thus doubles the contribution of the parent node to the sum over the nodes[7]. When we split a node in the jth coordinate direction, the *sum* of the lengths of the child nodes in the jth direction equals the length of the parent node in that direction; thus, the split does not affect the sum over the nodes. Hence, we have

$$\sum_{l=1}^{2^\ell} V_l' \leq 2M \cdot 2^{\ell - \frac{\ell}{d}} \tag{11}$$

when ℓ is a multiple of d. When ℓ is not a multiple of d, the last incomplete round may not contain a split in the jth coordinate direction, and thus

$$\sum_{l=1}^{2^\ell} V_l' \leq 2M \cdot 2^{\ell - \lfloor \frac{\ell}{d} \rfloor} \leq 4M \cdot 2^{\ell - \frac{\ell}{d}}.$$

□

[6] Since we are proving an upper bound it does not matter that we double count nodes that cross more than one $(d-1)$-face.

[7] Here we assume that the splitting hyperplane intersects with $[-M, M]^d$. If the splitting hyperplane does not intersect with $[-M, M]^d$, then the split does not increase $\sum_{l=1}^{2^\ell} V_l'$. Thus, the inequality in (11) holds as an equality if and only if all the splitting hyperplanes that are not orthogonal to the jth coordinate direction intersect with $[-M, M]^d$.

For completeness, we prove here the following well-known variation of Markov's inequality.

Lemma 3. *Suppose H is an event, $a > 0$, and X is a non-negative random variable for which $E|X| < \infty$. Then,*

$$P\{X > a, H\} \leq \frac{E[\mathbb{1}_H X]}{a}.$$

Proof. We can write

$$E[\mathbb{1}_H X] \geq E[\mathbb{1}_H X \mathbb{1}\{X > a\}] \geq aE[\mathbb{1}_H \mathbb{1}\{X > a\}] = aP\{X > a, H\},$$

and the result follows by dividing by a. □

Proof of Theorem 3

Proof. Choose a coordinate direction $j \in \{1, \ldots, d\}$ and denote the length of an edge of the hyperrectangle R_l in that direction by V_l. Since the coordinate direction was chosen arbitrarily, it suffices to show that $V_{q(X)}$ converges to zero in probability to prove that also the cell diameter converges to zero in probability.

For any training set size n, define $\ell' := \min(\ell, \lfloor \log_2 \log_2 n \rfloor)$. Since for any training set[8] D_n the probability $P\{V_{q(X)} > \delta \mid D_n\}$ is non-increasing w.r.t. to the tree height and $\ell' \leq \ell$, in order to prove that $P\{V_{q(X)} > \delta\} \to 0$ for the original tree height ℓ, it is sufficient to show that it goes to zero for the tree height ℓ'.

For any $\theta > 0$, there exists $M > 0$ s.t. $P\{X \notin [-M, M]^d\} < \theta$. The box $[-M, M]^d$ divides any partition of $\mathbb{R}^d$ into three disjoint sets A, B, and C as defined in (10) that correspond to the indexes of the nodes completely inside $[-M, M]^d$, the indexes of the nodes crossing its border and the indexes of the nodes completely outside of it, respectively[9]. We can now decompose the probability of the event $\{V_{q(X)} > \delta\}$ into three parts corresponding to the sets A, B, and C:

$$P\{V_{q(X)} > \delta\} \leq P\{V_{q(X)} > \delta, q(X) \in A\} + P\{q(X) \in B\} + P\{q(X) \in C\}. \tag{12}$$

Choose $\epsilon > 0$ and denote the event that no partition element has a probability mass larger than $(1+\epsilon)/2^{\ell'}$ by

$$G := \bigcap_{l=1}^{2^{\ell'}} \left\{ \mu(R_l) \leq \frac{1+\epsilon}{2^{\ell'}} \right\},$$

where $\mu(A) := P\{X \in A\}$ is the probability distribution of X for any measurable set A. Our strategy is to first handle this case where all the leafs contain an approximately equal probability mass, and then bound the probability of G^C by applying the Vapnik-Chervonenkis inequality to show the uniform convergence of the empirical distribution of X to its true distribution in the class of leafs of a chronological k-d tree, i.e., in the class of hyperrectangles in $\mathbb{R}^d$. To this end, we further partition the right hand side of (12) as

$$P\{V_{q(X)} > \delta\} \leq \underbrace{P\{V_{q(X)} > \delta, q(X) \in A, G\}}_{I} + \underbrace{P\{q(X) \in B, G\}}_{II} + \underbrace{P(G^C)}_{III} + \underbrace{P\{q(X) \in C\}}_{IV}$$

and bound these four terms.

Term IV: Since $P\{q(X) \in C\} \leq P\{X \notin [-M, M]^d\} < \theta$, we can get this term as small as desired by choosing a small enough θ.

Term I: By applying Lemma 3 (with the event $\{q(X) \in A\} \cap G$), we see that

$$P\{V_{q(X)} > \delta, q(X) \in A, G\} \leq \frac{E\left[\mathbb{1}_G V_{q(X)} \mathbb{1}\{q(X) \in A\}\right]}{\delta}.$$

[8] To keep notation consistent throughout the article, we denote the training set also here by $D_n := \{(X_i, Y_i)\}_{i=1}^n$. However, it should be observed that the chronological k-d tree uses only the inputs $X_1, \ldots, X_n$ to learn the partition; thus, the learned partition does not depend on the labels $Y_1, \ldots, Y_n$.

[9] We define the sets A, B, and C here for a tree of height ℓ'.

Therefore, it suffices to bound $E\left[\mathbb{1}_G V_{q(X)} \mathbb{1}\{q(X) \in A\}\right]$, for which we have

$$
\begin{aligned}
E\left[\mathbb{1}_G V_{q(X)} \mathbb{1}\{q(X) \in A\}\right] &= E\left[\mathbb{1}_G E\left[V_{q(X)} \mathbb{1}\{q(X) \in A\} \,|\, D_n\right]\right] \\
&= E\left[\mathbb{1}_G \sum_{l=1}^{2^{\ell'}} \mu(R_l) V_l \mathbb{1}\{l \in A\}\right] \\
&= E\left[\mathbb{1}_G \sum_{l \in A} \mu(R_l) V_l\right] \\
&\leq \frac{1+\epsilon}{2^{\ell'}} E\left[\sum_{l \in A} V_l\right] \\
&\leq \frac{1+\epsilon}{2^{\ell'}} \cdot 4M \cdot 2^{\ell' - \frac{\ell'}{d}} \\
&= 4M \cdot \frac{1+\epsilon}{2^{\ell'/d}},
\end{aligned}
\tag{13}
$$

where the outermost expectation on the right hand side is w.r.t. D_n, the first inequality follows from the definition of G, and the second inequality follows from Lemma 1. Since by assumption $\ell' \to \infty$ when $n \to \infty$, also $P\{V_{q(X)} > \delta, q(X) \in A, G\} \to 0$.

Term II: Applying a similar technique as in (13), we have

$$
\begin{aligned}
P\{q(X) \in B, G\} &= E\left[\mathbb{1}_G P\{q(X) \in B \,|\, D_n\}\right] \\
&= E\left[\mathbb{1}_G \sum_{l=1}^{2^{\ell'}} \mathbb{1}\{l \in B\} \mu(R_l)\right] \\
&\leq \frac{1+\epsilon}{2^{\ell'}} E\left[N_B\right] \\
&\leq \frac{1+\epsilon}{2^{\ell'}} \cdot 4d \cdot 2^{\ell' - \frac{\ell'}{d}} \\
&= 4d \cdot \frac{1+\epsilon}{2^{\ell'/d}},
\end{aligned}
$$

where the expectation is w.r.t. D_n, the first inequality follows from the definition of G and the second inequality follows from Lemma 2. Hence, also $P\{q(X) \in B, G\} \to 0$ when $n \to \infty$.

Term III: Finally, we bound the probability of the event G^C. Let $\mathcal{R}$ be the class of all hyperrectangles in $\mathbb{R}^d$. The Vapnik-Chervonenkis dimension of $\mathcal{R}$ is $2d$ (see, e.g., Theorem 13.8. by Devroye et al. (1996, p. 220-221)), and hence we have $s(\mathcal{R}, n) \leq n^{2d}$ for its shatter coefficient (see, e.g., Theorem 13.3. by Devroye et al. (1996, p. 218)). If $n \geq 2 \cdot \frac{\log_2 n}{\epsilon} \geq 2 \cdot \frac{2^{\ell'}}{\epsilon}$, then $\frac{1}{n} \leq \frac{1}{2} \cdot \frac{\epsilon}{2^{\ell'}}$. This means that

for large enough n, we have

$$
\begin{aligned}
P(G^C) &= P\left\{\exists l \text{ s.t. } \mu(R_l) > \frac{1+\epsilon}{2^{\ell'}}\right\} \\
&= P\left\{\exists l \text{ s.t. } \mu(R_l) - \left(\frac{1}{2^{\ell'}} + \frac{1}{n}\right) > \frac{\epsilon}{2^{\ell'}} - \frac{1}{n}\right\} \\
&\leq P\left\{\exists l \text{ s.t. } \mu(R_l) - \mu_n(R_l) > \frac{\epsilon}{2^{\ell'}} - \frac{1}{n}\right\} \\
&\leq P\left\{\exists l \text{ s.t. } \mu(R_l) - \mu_n(R_l) > \frac{\epsilon}{2^{\ell'+1}}\right\} \\
&\leq P\left\{\sup_{R\in\mathcal{R}} |\mu(R) - \mu_n(R)| > \frac{\epsilon}{2^{\ell'+1}}\right\} \\
&\leq 8s(\mathcal{R}, n)\exp\left\{-\frac{n\epsilon^2}{128\cdot 2^{2\ell'}}\right\} \\
&\leq 8n^{2d}\exp\left\{-\frac{n\epsilon^2}{128\cdot 2^{2\ell'}}\right\} \\
&\leq 8n^{2d}\exp\left\{-\frac{n\epsilon^2}{128(\log_2 n)^2}\right\} \to 0
\end{aligned}
\tag{14}
$$

when $n \to \infty$. The first inequality on the right hand side of (14) follows because for the empirical measure—denoted by $\mu_n(A) := \frac{1}{n}\sum_{i=1}^n \mathbb{1}_A(X_i)$ for any measurable set A—of any leaf R_l it holds that $\mu_n(R_l) \leq \frac{1}{2^{\ell'}} + \frac{1}{n}$. The fourth inequality follows from the Vapnik-Chervonenkis inequality (Vapnik and Chervonenkis, 1971); we use the version presented in Theorem 12.5. by Devroye et al. (1996, p. 197-198). The last inequality follows because $2^{2\ell'} \leq (\log_2 n)^2$ by the definition of ℓ'. □

B Experimental setup

All the algorithms, hyperparameter combinations, and code used in the experiments are included in a repository that is attached to the supplements.

B.1 Computing environment

The experiments were ran on a machine with two Xeon E5-2680 v4 2.4GHz processors, 256GB RAM and CentOS 7 as the operating system. All queries were ran using only a single thread. The algorithms and test code were written in C++14 and compiled using GCC 5.4.0 with the optimization flags `-Ofast` and `-march=native`.

B.2 Data sets

Table 2 contains the specifications of the data sets. We used four publicly available and commonly used benchmark data sets (Fashion[10], GIST[11], STL-10[12], and Trevi[13]) consisting of raw or preprocessed images. We randomly divided the original data sets into the corpus, the validation set ($n_{\text{validation}} = 1000$), and the test set ($n_{\text{test}} = 1000$). The corpus $\{c_i\}_{i=1}^m$ was used as a training set. Since in the previous benchmarks for ANN search (Aumüller et al., 2019; Li et al., 2019) the problem is not considered in the machine learning setting, they do not use a distinct test set, but present the optimal results on the validation set. To follow this standard practice, we also present the results on the validation set, but note that the results were stable between the validation and test sets; there was some random variability, but we observed no signs of overfitting to the validation set.

[10] https://github.com/zalandoresearch/fashion-mnist
[11] http://corpus-texmex.irisa.fr
[12] https://cs.stanford.edu/ acoates/stl10
[13] http://phototour.cs.washington.edu/patches

Table 2: Data sets used in the experiments

Data set	corpus size m	dimension d
Fashion	58000	784
GIST	100000	960
STL-10	98000	9216
Trevi	99120	4096

B.3 Hyperparameter settings

According to our initial experiments, the performance of each type of a tree was robust w.r.t. its sparsity/randomization parameter (the number of the uniformly at random chosen coordinate directions $a \in \{1, \ldots, d\}$ from which the optimal split direction was chosen for multiclass classification trees; the dimensionality a of the random subspace in which first principal component was approximated for PCA trees; the expected number a of the non-zero components in the random vectors onto which the node points are projected in RP trees; and the number o of the highest variance directions of the node points from which the split direction was chosen uniformly at random in k-d trees). Therefore, we kept these parameters fixed in the final experiments: for multiclass classification trees and the PCA trees we used the value $a = \lceil \sqrt{d} \rceil$; for the RP trees the value $a = 1/\sqrt{d}$; and for the k-d trees the value $o = 5$. Further, we set the learning rate of the iterative PCA algorithm in PCA trees to $\gamma = 0.01$ and the maximum number of iterations to $t = 20$.

For the recall levels on the range [0.5, 0.99] considered in the article, the optimal numbers of trees T were generally on the range [5, 200], the optimal depths of the trees were on the range [10, 15], and the optimal values of the threshold parameter τ were on the range [1, 20] for PCA, RP, and k-d trees that use the raw counts as score function values. For multiclass classification trees that use the probability estimates (4) to select the candidate set, the optimal values of the threshold parameter τ were on the range [0.00001, 0.005]. We observed that using a value $k' > k$ to learn the trees sometimes improved performance. We tested values $k' \in \{10, 50, 100\}$ for learning the trees, with $k' = 10$ or $k' = 50$ usually being the optimal parameter value when $k = 10$.

For the other algorithms, we used the same hyperparameters as in ANN-benchmarks[14] as a starting point, and in many cases used even larger grids to ensure that the optimal hyperparameter settings were found.

C Data structures

In this section we review the five types of trees considered in this article (see C.1–C.5 below). Random projection, PCA, k-d trees and chronological k-d trees have been widely used for ANN search (see, e.g., (Silpa-Anan and Hartley, 2008; Muja and Lowe, 2014; Dasgupta and Sinha, 2015; Jääsaari et al., 2019)), whereas the multiclass classification tree is a standard data structure for classification. For completeness, we include the full descriptions of the algorithms here.

We begin by motivating the natural classifier (6) from the point of view of the multilabel problem reductions (see Reddi et al. (2019); Menon et al. (2019)). First note that since the label set $L(X)$ is a deterministic function of the query point X—which means that the conditional label probabilities $\eta_1(x), \ldots, \eta_m(x)$ are all equal to either 0 or 1—the Bayes classifier for 0-1 loss is obtained by thresholding these label probabilities by any $\tau \in [0, 1)$. As a corollary, the same holds also for other less strict loss functions, such as precision, recall, and Hamming loss. This justifies following the common practice of estimating the conditional label probabilities by reducing the original multilabel classification problem into a series of binary or multiclass classification problems (Menon et al., 2019).

In the *pick-all-labels* (PAL) (Reddi et al., 2019) reduction, a separate multiclass observation is created from each positive label, whereas in in the *one-versus-all* (OVA) reduction, each of the m labels is modeled as an independent binary classification problem. In the case of ANN search the maximum likelihood estimates for the label probabilities under the PAL and OVA reductions—i.e., under the

[14] https://github.com/erikbern/ann-benchmarks/blob/master/algos.yaml

multinomial and binomial models, respectively—are $\hat{\theta}_{\mathrm{PAL}} = \frac{1}{nk}\sum_{i=1}^{n} y_{ij}$ and $\hat{\theta}_{\mathrm{OVA}} = \frac{1}{n}\sum_{i=1}^{n} y_{ij}$. Since the size of the label set $L(x)$ is always k, the two estimates are proportional to each other, $\hat{\theta}_{\mathrm{OVA}} = k\hat{\theta}_{\mathrm{PAL}}$, and hence, given the partition $\mathcal{P}$, the parameter estimates of the natural classifier (6)—i.e., observed label proportions among the training set points in a given partition element—minimise the log-likelihood under both reductions.

Motivated by the above, we use the natural classifier (6) for prediction in combination with all tree types. When an ensemble of trees is used as a classifier, we compute the conditional label probability estimates of the ensemble as in (5) by averaging the contributions of the individual trees.

C.1 Chronological k-d tree

The chronological k-d tree (Bentley, 1975) was the first data structure proposed for speeding up nearest neighbor search. It rotates the split directions and uses the same split direction for all the nodes at one level of a tree. At the first level the training data is split at the median of the first coordinates of the data points. At the second level both nodes are split at the median of the second coordinates of the node points. At the $(d+1)$th level, the nodes are split again at the median of the first coordinates, and so on (see Algorithm 1). More adaptive version of the k-d tree that splits at the coordinate direction in which the node points have the highest variance was proposed by Friedman et al. (1976); we use a randomized version of this adaptive k-d tree (see Sec. C.3 and Algorithm 3) in the experiments of this article.

Algorithm 1 Grow a chronological k-d tree (Bentley, 1975)

Input: a set of node points $X \subset \{x_1, \ldots, x_n\}$, current level ℓ', maximum height ℓ
Output: a node of a tree
procedure GROW-KD(X, ℓ', ℓ)
    if $\ell' = \ell$ **then**
        return X node as a leaf node
    $\hat{r} \leftarrow (\ell' \text{ modulo } d) + 1$
    $\hat{s} \leftarrow$ median of the $\hat{r}$th coordinates of the node points
    left $\leftarrow$ GROW-KD($\{x_i \in X \,:\, x_{i\hat{r}} \leq \hat{s}\}$, $\ell'+1$, ℓ)
    right $\leftarrow$ GROW-KD($\{x_i \in X \,:\, x_{i\hat{r}} > \hat{s}\}$, $\ell'+1$, ℓ)
    return $(\text{left}, \text{right}, \hat{r}, \hat{s})$ as an inner node

C.2 Multiclass classification tree

As discussed at the beginning of the section, the maximum likelihood estimates of the piecewise constant model under both the PAL and OVA reductions coincide in the special case of ANN search. Thus, in principle it would make no difference which one of these reductions we employed to learn the classification trees. However, in practice computation of the binomial likelihood requires keeping track of the contributions of the negative labels which is inconvenient when the label space is large. Thus, we employ the PAL reduction where each positive label is modeled by a separate multiclass observation, and learn the standard multiclass classification trees by greedily maximising the multinomial log-likelihood (i.e., use the multiclass cross-entropy as a split criterion); see Algorithm 2 for details.

To grow a random forest, we randomize the multiclass classification trees by optimising the split point only in randomly chosen $a = \lceil \sqrt{d} \rceil$ coordinate directions at each node of a tree. We do not use bootstrap samples, but fit each tree to the original training data. To decrease learning time, we subsample 100 training points at each node (if node size > 100), and use only this subset to optimize the splits; we did not observe any negative impact on prediction performance.

C.3 k-d tree

Algorithm 3 details the recursive algorithm for growing a randomized k-d tree. As in multiclass classification trees, the splits are restricted to the directions of the coordinate axes. In k-d trees (Friedman et al., 1976) the normal of the splitting hyperplane is chosen as the coordinate direction $r \in \{1, \ldots, d\}$ along which the node points have the highest variance. The split point $\hat{s}$ is chosen as the median of

Algorithm 2 Grow a randomized multiclass classification tree

Input: a set of node points $X \subset \{x_1, \ldots, x_n\}$, current depth, maximum depth ℓ, sparsity parameter a
Output: a node of a tree
procedure GROW(X, depth, $\ell, a = \lceil \sqrt{d} \rceil$)
  if depth $= \ell$ **then**
    return X as a leaf node
  $D \leftarrow a$ random unique dimensions from $\{1, \ldots, d\}$
  $(\text{maxgain}, \hat{r}, \hat{s}) \leftarrow (0, 0, 0)$
  $N \leftarrow |X|$
  for $r \in D$ **do**
    let $s_1 \leq s_2 \leq \cdots \leq s_N$ be the rth coordinate of points $x \in X$ sorted in ascending order
    for $s \in \{s_1, \ldots, s_N\}$ **do**
      $X_{\text{left}} \leftarrow \{x_i \in X : x_{ir} \leq s\}$; $N_{\text{left}} = |X_{\text{left}}|$
      $X_{\text{right}} \leftarrow \{x_i \in X : x_{ir} > s\}$; $N_{\text{right}} = |X_{\text{right}}|$
      for $j \in \{1, \ldots, m\}$ **do**
        $v_j^{(\text{left})} \leftarrow \sum_{x_i \in X_{\text{left}}} y_{ij}$
        $v_j^{(\text{right})} \leftarrow \sum_{x_i \in X_{\text{right}}} y_{ij}$
        $v_j \leftarrow \sum_{x_i \in X} y_{ij}$
        $\hat{\theta}_j^{(\text{left})} := v_j^{(\text{left})} / (kN_{\text{left}})$
        $\hat{\theta}_j^{(\text{right})} := v_j^{(\text{right})} / (kN_{\text{right}})$
        $\hat{\theta}_j := v_j / (kN)$
      gain $\leftarrow \sum_{j=1}^m v_j^{(\text{left})} \log \hat{\theta}_j^{(\text{left})} + \sum_{j=1}^m v_j^{(\text{right})} \log \hat{\theta}_j^{(\text{right})} - \sum_{j=1}^m v_j \log \hat{\theta}_j$
      if gain $>$ maxgain **then**
        $(\text{maxgain}, \hat{r}, \hat{s}) \leftarrow (\text{gain}, r, s)$
  if maxgain ≤ 0 **then**
    return X as a leaf node
  left $\leftarrow$ GROW($\{x_i \in X : x_{i\hat{r}} \leq \hat{s}\}$, depth + 1, ℓ, a)
  right $\leftarrow$ GROW($\{x_i \in X : x_{i\hat{r}} > \hat{s}\}$, depth + 1 ℓ, a)
  return $(\text{left}, \text{right}, \hat{r}, \hat{s})$ as an inner node

Algorithm 3 Grow a randomized k-d tree (Silpa-Anan and Hartley, 2008)

Input: a set of node points $X \subset \{x_1, \ldots, x_n\}$, current level, maximum height ℓ, the number of highest variances directions from which the split direction is sampled o
Output: a node of a tree
procedure GROW-KD(X, level, $\ell, o = 5$)
  if level $= \ell$ **then**
    return X as a leaf node
  $D \leftarrow$ set of o coordinate directions in which the node points have the highest variance
  $\hat{r} \leftarrow$ uniformly at random sampled dimension from the set D
  $\hat{s} \leftarrow$ median of $\hat{r}$th coordinates of the node points
  left $\leftarrow$ GROW-KD($\{x_i \in X : x_{i\hat{r}} \leq \hat{s}\}$, level + 1, ℓ, o)
  right $\leftarrow$ GROW-KD($\{x_i \in X : x_{i\hat{r}} > \hat{s}\}$, level + 1 ℓ, o)
  return $(\text{left}, \text{right}, \hat{r}, \hat{s})$ as an inner node

the $\hat{r}$th coordinate of the node points. To grow an ensemble of randomized trees, we use the randomization scheme proposed by Silpa-Anan and Hartley (2008): instead of splitting at the direction of the highest variance, we draw uniformly at random one of the o highest variance directions and use it as a split direction $\hat{r}$. We use the default value $o = 5$ recommended by Muja and Lowe (2014) for this hyperparameter.

C.4 PCA tree

In a PCA tree (Sproull, 1991), the projection direction at each node is the first principal component, i.e. the direction the node points have the greatest variance when projected onto. PCA trees are on the one hand slow to compute, as computing exact PCA is expensive, and on the other hand they are deterministic and thus multiple trees cannot be grown to boost accuracy.

To solve the first problem, McCartin-Lim et al. (2012) proposed an *approximate PCA tree*, which uses gradient descent updates to approximate the first principal component of the data at each node of the tree. To address the second problem, Jääsaari et al. (2019) proposed a *sparse approximate PCA tree*, which draws at each node of the tree uniformly at random only $a = \sqrt{d}$ dimensions, and computes the approximate first principal component in the subspace defined by these dimensions.

The gradient descent update for approximate PCA is

$$r_t := r_{t-1} + \gamma \operatorname{Cov}(Z) r_{t-1}, \qquad r_t := r_t / \left\| r_t \right\|_2 ,$$

where r_t is the projection vector at time t, Z is a matrix containing the data, and γ is the learning rate which we fix as 0.01. We did not observe further tuning of this hyperparameter to be necessary.

Algorithm 4 details a recursive algorithm for growing a sparse approximate PCA tree. Algorithm 5 details the actual approximate PCA algorithm used to find the split direction. On line 7, the matrix Z is formed by taking the vectors of the current points X as columns of a matrix and then slicing only the rows (dimensions) that were randomly selected into the set D on line 6. The sample covariance matrix C is formed from Z on lines 8-9. Lines 10-11 initialize the projection vector from the unit sphere, while lines 12-17 implement the gradient descent update described above. By default, we do $t = 20$ iterations of gradient descent, unless the ℓ_1 norm of the projection vector changes by less than $\epsilon = 0.01$ in a single iteration.

Algorithm 4 Grow a randomized PCA tree (Jääsaari et al., 2019)

1: **Input:** a set of node points $X \subset \{x_1, \ldots, x_n\}$, current depth, maximum depth ℓ, sparsity parameter a, maximum number of iterations t, learning rate γ, threshold parameter ϵ
2: **Output:** a node of a tree
3: **procedure** GROW(X, depth, $\ell, a = \lceil \sqrt{d} \rceil, t = 20, \gamma = 0.01, \epsilon = 0.01$)
4: **if** depth $= \ell$ **then**
5: **return** X as a leaf node
6: direction $\leftarrow$ PCA-GENERATE-SPLIT-DIRECTION($X, a, t, \gamma, \epsilon$)
7: proj $\leftarrow$ PROJECT(X, direction)
8: $\hat{s} \leftarrow$ MEDIAN(proj)
9: $X_{\text{left}} \leftarrow$ points in X for which $\text{proj} \leq \hat{s}$
10: $X_{\text{right}} \leftarrow$ points in X for which $\text{proj} > \hat{s}$
11: left $\leftarrow$ GROW(X_{left}, depth + 1, $\ell, a, t, \gamma, \epsilon$)
12: right $\leftarrow$ GROW(X_{right}, depth + 1, $\ell, a, t, \gamma, \epsilon$)
13: **return** (left, right, direction, $\hat{s}$) as an inner node

C.5 Random projection tree

Algorithm 6 describes a process of growing a sparse random projection (RP) tree. In RP trees (Dasgupta and Freund, 2008; Dasgupta and Sinha, 2015), the normal of the splitting hyperplane is chosen uniformly at random from d-dimensional standard normal distribution $N(0, I)$. The node points are projected into this random vector, and the node is split at the median of the projections. Hyvönen et al. (2016) proposed a sparse variant, where the components of the random vectors are generated from the standard normal distribution with the probability a, and are zero with the probability $1 - a$ (see Algorithm 7). For this hyperparameter we use the default value $a = 1/\sqrt{d}$ as recommended by Hyvönen et al. (2016). This decreases both the index construction time and the query routing time by a factor of $\sqrt{d}$ compared to the original dense RP trees.

Algorithm 5 Generate projection vector for a randomized PCA tree (Jääsaari et al., 2019)

Input: a set of node points $X \subset \{x_1, \ldots, x_n\}$, sparsity parameter a, maximum number of iterations t, learning rate γ, threshold parameter ϵ
Output: an approximate first principal component p in a-dimensional subspace; observe that p has only a nonzero components.
procedure PCA-GENERATE-SPLIT-DIRECTION($X, a, t, \gamma, \epsilon$)
    $N \leftarrow |X|$
    initialize d-dimensional vector p with zeros
    $D \leftarrow a$ random unique dimensions from $1, \ldots, d$
    $Z \leftarrow$ points in X with all components but those in D removed
    $M \leftarrow Z(I_N - \frac{1}{N}\mathbf{1}\mathbf{1}^{\mathrm{T}})Z^T$
    $C \leftarrow \frac{1}{N-1}MM^T$
    $r \leftarrow \mathcal{X}_a(\mathbf{0}, \mathbf{I})$
    $r \leftarrow r/\|r\|_2$
    for $i \in \{1, \ldots, t\}$ **do**
        $r' \leftarrow r$
        $r \leftarrow r + \gamma Cr$
        $r \leftarrow r/\|r\|_2$
        if $\|r - r'\|_1 < \epsilon$ **then**
            break
    $j \leftarrow 1$
    for $i \in D$ **do**
        $p[i] \leftarrow r[j]$
        $j \leftarrow j + 1$
    return p

Algorithm 6 Grow a sparse RP tree (Hyvönen et al., 2016)

Input: set of node points $X \subset \{x_1, \ldots, x_n\}$, current depth, maximum depth ℓ, sparsity parameter a
Output: node of a tree
procedure GROW-RP(X, depth, $\ell, a = 1/\sqrt{d}$)
    if depth $= \ell$ **then**
        return X as a leaf node
    direction $\leftarrow$ RP-GENERATE-SPLIT-DIRECTION(a)
    proj $\leftarrow$ PROJECT(X, direction)
    $\hat{s} \leftarrow$ MEDIAN(proj)
    $X_{\text{left}} \leftarrow$ points in X for which $\text{proj} \leq \hat{s}$
    $X_{\text{right}} \leftarrow$ points in X for which $\text{proj} > \hat{s}$
    left $\leftarrow$ GROW-RP(X_{left}, depth + 1, ℓ, a)
    right $\leftarrow$ GROW-RP(X_{right}, depth + 1, ℓ, a)
    return $(\text{left}, \text{right}, \text{direction}, \hat{s})$ as an inner node

Algorithm 7 Generate a normal of the splitting hyperplane for a sparse RP tree (Hyvönen et al., 2016)

Input: sparsity parameter a that is the expected proportion of non-zero components in the output vector
Output: a random d-dimensional vector
procedure RP-GENERATE-SPLIT-DIRECTION(a)
    initialize d-dimensional vector p with zeros
    for $i \in \{1, \ldots, d\}$ **do**
        if RANDOM(0, 1) $\leq a$ **then**
            generate $p[i] \sim N(0, 1)$
    return p

D Additional experimental results

D.1 Index construction times

The index construction times of the algorithms at the optimal parameters for the recall levels $R = 0.8, 0.9, 0.95$ are shown in Table 3. The computation time for finding the nearest neighbors (i.e., the labels) of the corpus points is not included in the index construction times, since they are computed only once for each data set; they are found in Table 5 (in the column "exact").

The ensembles of unsupervised trees are relatively fast to build, especially on the high-dimensional data sets: on STL-10, PCA trees have the fastest query times, and are an order of magnitude faster to train compared to the graph and quantization methods.

The multiclass classification trees (RF) are slower to train compared to the unsupervised (PCA, KD, and RP) trees. We expect that their training times could be decreased by standard techniques, such as using weighted quantile sketches (Greenwald and Khanna, 2001; Zhang and Wang, 2007; Chen and Guestrin, 2016) when optimizing the split points.

Table 3: Index building time (seconds) at optimal parameters. The fastest time for each recall level is typeset in bold.

data set	R (%)	PCA	KD	RP	RF	ANNOY	HNSW	IVF-PQ
Fashion	80	2.014	**0.929**	1.298	14.422	1.244	1.518	5.867
	90	1.621	1.500	**1.284**	25.591	11.564	1.690	5.867
	95	1.847	2.208	1.925	45.683	11.564	**1.518**	5.867
GIST	80	27.732	27.162	30.520	131.430	16.349	19.114	**13.031**
	90	30.313	36.664	56.624	300.340	62.931	21.560	**13.031**
	95	30.313	49.707	48.056	300.340	**8.319**	26.456	13.031
STL-10	80	**4.497**	25.426	12.204	316.790	32.036	93.393	92.577
	90	**8.891**	30.814	12.145	647.320	489.482	132.500	92.577
	95	**7.918**	28.286	12.145	466.520	489.480	201.070	92.577
Trevi	80	**4.900**	11.158	10.185	420.250	141.794	60.044	43.520
	90	18.937	13.886	**10.169**	420.250	141.790	60.044	43.520
	95	18.937	12.432	**11.261**	420.250	141.790	60.044	43.520

D.2 Comparison to graph and quantization methods

To empirically justify studying partition-based ANN algorithms, we also include in the comparison Hierarchical Navigable Small World (HNSW) (Malkov and Yashunin, 2018) graphs and Inverted File Product Quantization (IVF-PQ) Jegou et al. (2010), that were the fastest graph-based and the fastest quantization-based algorithm, respectively, according to ANN-benchmarks (Aumüller et al., 2019) project at the time of its publication[15]. For completeness, we also include the commonly-used tree-based method ANNOY[16] in the comparison. See Table 1 for the results. We emphasize that this is not a benchmark article with the goal of proposing a single ANN algorithm and demonstrating its superiority over the competition—rather, we aim to establish a widely applicable theoretical framework for partition-based ANN search.

D.3 Comparison of candidate set selection methods: all data sets

Figure 2 contains the results of the comparison between the candidate set selection for all the four data sets. The results validate the theoretical findings: using space-partitioning trees as natural classifiers to select the candidate set improves their performance consistently for all the tree types and

[15] As of May 2022, the fastest graph-based method is NGTQG (`https://github.com/yahoojapan/NGT/blob/master/bin/ngtqg/README.md`), the fastest quantization-based algorithm is SCANN (Guo et al., 2020) (see `http://ann-benchmarks.com/index.html` for updated results).

[16] `https://github.com/spotify/annoy`

Table 4: Query times (seconds / 1000 queries) at different recall levels for the different tree types. The fastest method in each case is typeset in boldface.

data set	R (%)	PCA	KD	RP	RF	ANNOY	HNSW	IVF-PQ
Fashion	80	0.075	0.076	0.099	**0.063**	0.193	0.064	0.266
	90	0.111	0.126	0.172	0.095	0.296	**0.089**	0.291
	95	0.163	0.171	0.261	0.146	0.419	**0.097**	0.340
GIST	80	1.330	0.958	1.009	0.705	2.525	**0.524**	0.872
	90	2.942	2.286	2.226	1.530	5.973	**0.819**	2.037
	95	5.641	4.451	4.598	3.253	7.477	**1.212**	2.657
STL-10	80	**0.382**	0.872	1.211	0.756	21.110	1.473	6.140
	90	**0.756**	2.126	3.248	1.774	24.826	2.717	6.860
	95	**1.315**	4.376	7.330	3.654	35.459	3.963	6.860
Trevi	80	**0.330**	0.543	0.591	0.582	5.259	0.705	1.677
	90	**0.684**	1.464	1.468	1.234	9.921	1.202	1.892
	95	**1.212**	3.244	3.289	2.350	17.172	1.896	2.655

data sets compared to retrieving the candidate set in the lookup-based paradigm (lookup search and voting). Voting also always outperforms lookup search—this is not surprising since lookup search is a special case of voting with $\tau = 0$ as discussed in Sec. 5. For completeness, we also include the corresponding special case of the natural classifier with $\tau = 0$ in the comparison.

D.4 Training classifiers with noisy labels

The disadvantage of the supervised ANN search algorithms compared to the purely unsupervised algorithms is that they require computing the true nearest neighbors $\{y_i\}_{i=1}^n$ of the training set points $\{x_i\}_{i=1}^n$, which is an $\mathcal{O}(nmd)$ operation. This is not a problem for the benchmark data sets used in this article—for the largest data set (STL-10, $n = 98000$, $m = 98000$, $d = 9216$), computing the ground truth took 50 minutes on a single machine—but in the typical applications of ANN search the corpus size may be hundreds of millions or even billions.

The labels used to train the classifier do not have to be exact. We can also compute the *approximate* nearest neighbors of the training set $\{x_i\}_{i=1}^m$ and use them as labels $\{y_i\}_{i=1}^m$ to train the classifier. For instance, using approximate nearest neighbors computed at average recall level of 90% amounts to using noisy labels with 10% noise.

To test how the noisy labels affect the performance, we fit the random forest to training sets with 10%, 20%, 30%, 40%, and 50% label noise. The noisy labels are obtained by running the MRPT algorithm (Hyvönen et al., 2016) (i.e., an ensemble of RP trees where the candidate set is selected by voting) in combination with the automatic hyperparameter tuning algorithm (Jääsaari et al., 2019) to find the approximate nearest neighbors of the training set points with recall levels 90%, 80%, 70%, 60%, and 50%, respectively.

The results (c.f. Fig 3) indicate that tree-based classifiers are robust with respect to label noise: training the random forest with 10% label noise has no visible effect on the performance of the algorithm, and even training on labels with 50% noise has very little effect.

Computing times for exact and approximate nearest neighbors for the training set are found on Table 5 for all the four data sets. The results indicate that significant savings in preprocessing time can be obtained by using noisy labels: for instance, on STL-10 computing the exact computation took 50 minutes, whereas the approximate computation took only 1-10 minutes depending on the recall level.

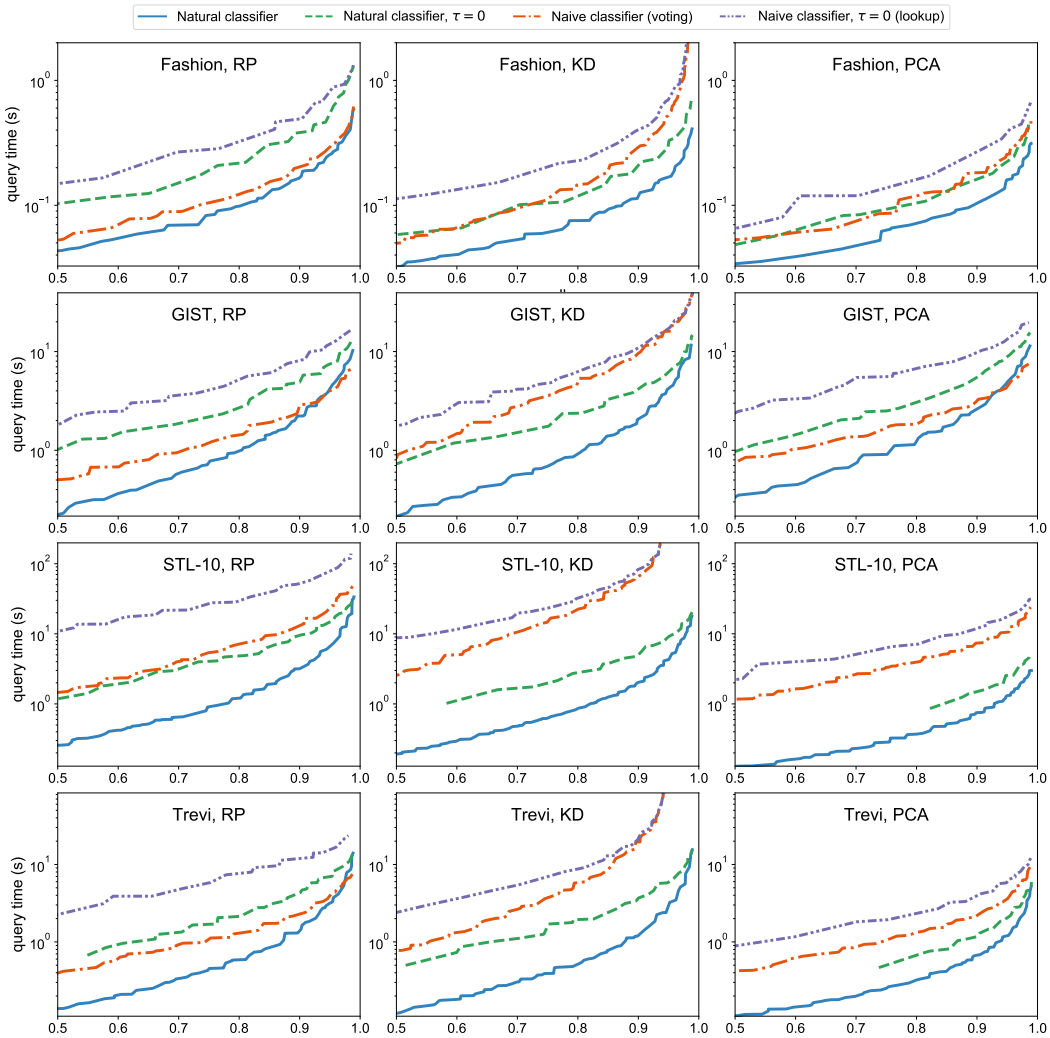

Figure 2: Recall vs. query time (log scale) of ensembles of, RP, k-d, and PCA trees. The solid blue line is the natural classifier proposed in this paper; the dash-dotted red line is the natural classifier with $\tau = 0$ that is included for completeness; the dashed green line is voting; and the double-dash-dotted violet line is lookup search. The natural classifier is the fastest and the lookup search is the slowest of the methods for each tree type.

Table 5: Computation times for exact (brute force) and noisy (MRPT algorithm) labels in seconds. The percentage is the average number of correct approximate nearest neighbors.

data set	exact	95%	90%	70%	50%
Fashion	105.8	5.6	3.1	1.8	1.0
GIST	371.7	92.7	61.9	20.2	10.0
Trevi	1450.2	118.6	104.4	31.4	24.1
STL-10	2992.7	596.3	409.0	97.2	66.5

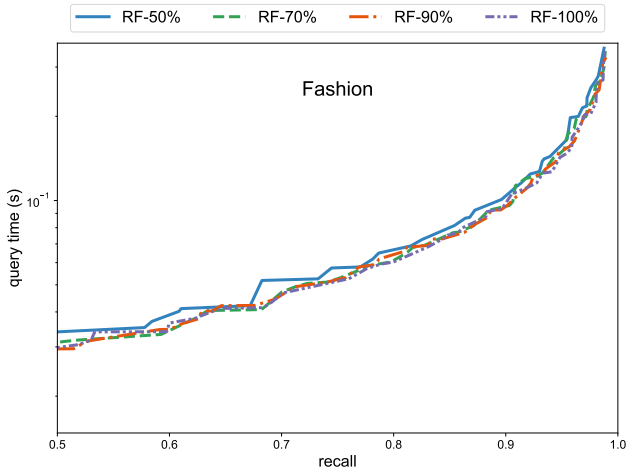

Figure 3: Recall vs. query time (log scale) of a random forest trained with different amounts of label noise on the Fashion data set. RF-100% is the random forest trained using the exact k-nn matrix, RF-90% is the random forest trained using approximate k-nn matrix with that contains on average 90% of the correct neighbors, etc. Allowing 10% noise in labels has no visible effect on performance, and even allowing 50% noise has very little effect.